Why R Might Not Be the Best Choice for Parallel Programming

Lorenzo Benedetti
lorenzo.benedetti5@edu.unifi.it

Abstract

This paper presents a comparative analysis of sequential and parallel implementations of the K-means clustering algorithm using the R programming language. Parallelization in R was achieved through the use of the `parallel` *and* `doParallel` *packages. The study evaluates performance across different dataset sizes, number of clusters, and feature dimensionalities. While parallel R implementations exhibit moderate speedup over their sequential counterparts, overall performance gains are limited. In particular, execution times and scalability remain significantly inferior when compared to analogous implementations in C++ with OpenMP. The analysis highlights the challenges and limitations of achieving high-performance parallel computing in R, emphasizing the gap between R-based and lower-level language implementations in computational efficiency.*

Future Distribution Permission

1. Introduction

The increasing availability of large datasets and the need for efficient data analysis tools have made clustering algorithms, particularly K-means, fundamental in various fields such as machine learning, bioinformatics, and market segmentation. Although conceptually simple and widely adopted, the K-means algorithm can be computationally expensive when applied to high-dimensional or large-scale datasets. Addressing this limitation often involves leveraging parallel computing techniques to accelerate execution.

This study investigates the performance of K-means clustering implemented in the R programming language, comparing sequential and parallel approaches. Parallelization in R is achieved through the use of high-level packages such as `parallel` and `doParallel`, which offer a relatively straightforward interface for exploiting multicore CPU architectures.

The primary goal of this work is to assess the extent to which R-based parallelism can improve the performance of K-means, and how it compares to lower-level implementations. To this end, we conduct a performance comparison with an optimized C++ version of K-means using OpenMP for parallelization. This comparison highlights the gap in efficiency between high-level and low-level programming environments when it comes to parallel computation.

The key contributions of this work are:

- Implementation and evaluation of sequential and parallel versions of K-means clustering in R using the `parallel` and `doParallel` packages
- Empirical analysis of execution times and speedup across varying dataset sizes, number of clusters, number of threads and feature dimensionalities
- Comparative discussion of R's parallelization capabilities versus a C++ OpenMP implementation, with a focus on performance limitations

2. K-Means

2.1. Description

K-means clustering is an unsupervised learning algorithm that aims to partition n observations into k clusters, where each observation belongs to the cluster with the nearest mean (centroid). The algorithm operates in multidimensional space, making it suitable for a wide range of applications including image processing, customer segmentation, and pattern recognition. The algorithm's objective is to minimize the within-cluster sum of squares (WCSS), also known as the inertia. Mathematically, given a set of observations $(x_1, x_2, ..., x_n)$, where each observation is a d-dimensional real vector, k-means clustering aims to partition the n observations into k sets $S = S_1, S_2, ..., S_k$ to minimize:

$$\sum_{i=1}^{k} \sum_{x \in S_i} |x - \mu_i|^2 \tag{1}$$

where μ_i is the mean of points in S_i.

2.2. Algorithm

The K-means algorithm follows an iterative refinement approach:

```
Algorithm 1: K-Means Clustering
Input:
  • Dataset $X = \{x_0, \ldots, x_{\text{N\_SAMPLES-1}}\}$
  • N_SAMPLES
  • N_FEATURE_LIST list of dimensions
  • N_CLUSTERS
  • MAX_ITER number of epochs
  • NUM_THREADS list of threads
Output: Set of N_CLUSTERS
Assing each observation to the cluster with the
nearest centroid Calculate the Euclidean distance
repeat
    Calculate the new centroid of each cluster;
    Update the mean for each cluster;
until MAX_ITER is reached;
```

The algorithm is guaranteed to converge to a local optimum, though this may not be the global optimum. The quality of the final clustering depends significantly on the initial centroid selection, the number of clusters k and the number of epochs.

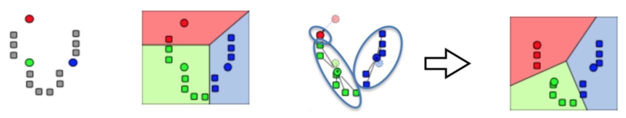

Figure 1. K-Means visualization

3. Implementation Details

3.1. Array of Structures (AoS)

In this implementation, all data structures were designed using the *Array of Structures* (AoS) approach rather than the *Structure of Arrays* (SoA) pattern. This decision was driven by the practical limitations of R, particularly its performance constraints when handling very large datasets. Due to the relatively low number of samples that can be effectively processed in R without incurring prohibitive execution times, the memory access patterns and cache locality optimizations offered by SoA are less beneficial. On the contrary, AoS offers a more straightforward implementation and tends to be more efficient in low-to-moderate data volumes, making it a more suitable choice for our experimental setup.

3.2. Compute Distance

The core computational step in K-means is the distance calculation between each data point and all centroids. We use the squared Euclidean distance as the similarity metric, computed as the sum of squared differences across all feature dimensions. This computation is performed explicitly using nested loops in both sequential and parallel implementations to maintain full control over memory usage and iteration order. While R offers vectorized alternatives, we avoided them due to their higher memory overhead and lower control over computational granularity in a parallel context.

3.3. Sequential K-Means

The sequential version of the K-means algorithm was implemented in base R without relying on external packages. The algorithm begins with the random initialization of centroids selected from the dataset. Each iteration consists of two main steps: assigning each point to the nearest centroid, and then recomputing centroids as the mean of all assigned points. Convergence is driven by a fixed number of iterations to ensure consistency in experimental results.

3.4. Parallel K-Means

The parallel implementation leverages the `doParallel` and `foreach` packages in R to distribute the workload of the assignment step across multiple CPU cores. The dataset is partitioned into chunks, each processed in parallel to determine the closest centroids. Results are then combined using `.combine = c`. The update step remains sequential due to the relatively small number of centroids, which does not justify the overhead of parallel reduction. The use of high-level parallel frameworks introduces non-negligible overhead and limits the attainable speedup, particularly when compared to more efficient implementations in low-level languages such as C++ with OpenMP.

```
# Process each chunk in parallel
results <- foreach(chunk = 1:n_cores, .combine =
    c) \%dopar\% {
  start_idx <- (chunk - 1) * chunk_size + 1
  end_idx <- min(chunk * chunk_size, n)

  if (start_idx > n) return(numeric(0))

  chunk_assignments <- rep(0, end_idx - start_idx
    + 1)

  for (i in start_idx:end_idx) {
    min_dist <- Inf
    best_cluster <- 0

    for (j in 1:k) {
      dist <- sum((x[i, ] - centroids[j, ])^2)
      if (dist < min_dist) {
        min_dist <- dist
        best_cluster <- j
      }
    }
    chunk_assignments[i - start_idx + 1] <- best_
    cluster
  }

  return(chunk_assignments)
}
```

`%dopar%` è un operatore del pacchetto `foreach` utilizzato per eseguire operazioni in parallelo.

3.5. Dataset Generation

Synthetic datasets were generated using Gaussian distributions centered around randomly initialized cluster centers. The `generate_dataset` function creates a data matrix with a user-specified number of samples, features, and clusters, ensuring reproducibility through fixed random seeds. Each sample is assigned to one of the clusters, and its feature vector is generated by adding Gaussian noise to the corresponding cluster center. The datasets are saved to CSV files for subsequent loading and processing, enabling consistent benchmarking across multiple experimental runs.

```
generate_dataset <- function(n_samples,
    n_features, n_clusters, random_state = 42) {
  set.seed(random_state)

  # Create cluster centers randomly
  centers <- matrix(rnorm(n_features * n_clusters
    ), nrow = n_clusters)

  # Assign samples to clusters
  y <- sample(1:n_clusters, n_samples, replace =
    TRUE)

  # Generate data points around cluster centers
  x <- matrix(0, nrow = n_samples, ncol =
    n_features)
  for (i in 1:n_samples) {
    cluster_center <- centers[y[i], ]
    x[i, ] <- cluster_center + rnorm(n_features,
    sd = 1.0)
  }

  return(list(x = x, y = y))
}
```

3.5.1 Experiment Execution and Visualization

The Python framework automates the execution of experiments across different thread counts:

- 1, 2, and 4 threads on a M1 MacBook Air
- 1, 2, 4, 8, 16, 32, 48 threads on an AMD Ryzen Threadripper 2970wx

The experiments are executed with both sequential and parallel implementation variants:

1. Executes the R implementation using subprocess:

```python
# Prepare arguments for R script
args = json.dumps({
  "dataset_file": input_file,
  "n_clusters": k,
  "max_iter": max_iter,
  "data_dir": str(DATA_DIR),
  "n_threads": n_threads
}).replace('"', '\\"')

# Call R function
r_cmd = [
  "Rscript", "-e",
  f"source('kmeans.R'); args <- fromJSON('{
    args}'); "
  f"result <- run_experiment(args); "
  f"cat(result)"
]
```

2. Collects timing data (measured using `Sys.time()`) and generates two types of visualizations using matplotlib:

   - Speedup comparison plots showing the relative performance improvement with increasing thread count
   - Execution time comparisons between sequential and parallel implementations

All results and visualizations are automatically saved in a structured directory for further analysis.

4. Experimental Results

This study primarily focuses on speedup as the key performance metric while also considering raw execution times to provide a comprehensive performance analysis. The speedup is calculated as the ratio between sequential and parallel execution times.

4.1. Performance Metrics

Our experimental evaluation encompasses several dimensions of analysis:

- Dataset dimensionality (varying from 2D to high-dimensional data)
- Number of data points (scaling from small to very large datasets)
- Number of clusters (varying k-means parameter)
- Thread count impact (scaling from single-thread to many-thread execution)

The experiments were conducted on two distinct hardware platforms to evaluate both consumer-grade and mid to high-performance scenarios:

- Apple M1 MacBook Air: Used for baseline performance testing, utilizing 4 performance cores. While the M1 chip features 8 cores in total (4 performance + 4 efficiency), we deliberately restricted our tests to the performance cores as the efficiency cores' significantly lower clock speeds would skew the parallel processing results.
- AMD Ryzen Threadripper 2970WX: Employed for scalability testing, featuring 24 cores and 48 threads, allowing us to analyze the implementation behavior under high thread counts. This platform enables investigation of potential scalability limitations and threading overhead effects.

For each analysis, we measure and compare:

- Sequential execution time (T_{seq})
- Parallel execution time (T_{par})
- Speedup ratio ($S = T_{seq}/T_{par}$)

All measurements are averaged over multiple runs (10) to ensure statistical significance, tested under identical conditions. The following sections present detailed analyses of these measurements, with particular emphasis on the scalability characteristics across different problem dimensions.

4.2. Speedup Analysis

This section presents a detailed analysis of the speedup achieved across various experimental dimensions using R parallelization. All experiments were conducted with a maximum of 30 iterations to ensure convergence. Unless otherwise specified, tests were performed using 100.000 data points, 32 clusters, and 2 dimensions.

4.2.1 Impact of Data Points

Although data scaling in R is decent, the performance remains significantly lower compared to the parallel implementation of K-means in C++ with OpenMP. In fact, while the C++ version was able to handle datasets with up to 100 million points without issues, the R implementation had to be stopped at just 1 million points due to excessively long execution times (over 30 minutes).

The table 1 below shows the speedup obtained by varying the number of data points from 1,000 to 1 million, using 16 clusters:

Points \ Threads	1	2	4
1'000	0.626	0.855	0.986
10'000	0.962	1.668	2.567
100'000	0.963	1.819	**2.864**
1'000'000	0.994	1.478	2.131

Table 1. Speedup implementation in R with varying number of points and threads.

Points \ Threads	1	2	4
1'000	0.831	0.862	0.526
10'000	0.805	1.276	1.626
100'000	0.843	1.607	1.803
1'000'000	0.854	1.654	3.070
10'000'000	0.820	1.636	3.163
100'000'000	0.849	1.674	**3.240**

Table 2. Speedup for SoA implementation in C++ with OpenMP with varying number of points and threads.

- For small datasets (1,000 points), the speedup is already close to 1, suggesting that the overhead of parallel execution is lower than the OpenMP's one.

- As the number of points increases, speedup improves, indicating effective parallelization — although not as much as expected.

- Surprisingly, the speedup at 1 million points is lower than at 100,000 points, which suggests that the R implementation may not scale efficiently for larger datasets.

- These results highlight the limitations of R in handling high-performance parallel computing tasks, especially when compared to the more scalable C++ + OpenMP implementation as showed in the table 2.

4.2.2 Impact of Dimensionality

Dataset dimensionality has a notable effect on the performance of the R parallel implementation. The results demonstrate good scalability as dimensionality increases, with speedup improving up to 2.721× at 32 dimensions. However, these values remain significantly below the maximum speedup of 3.689× observed with the C++ and OpenMP implementation.

Dim \ Threads	1	2	4
2D	0.989	1.489	2.297
3D	1.037	1.641	1.919
4D	1.033	1.656	2.311
8D	0.970	1.753	2.621
16D	1.027	1.334	2.629
32D	0.919	1.456	**2.721**

Table 3. Speedup with varying dimensionality.

- Speedup tends to increase with dimensionality, indicating that R's parallel implementation benefits from more computationally intensive tasks as dimensions grow.

- The highest speedup of 2.721× is achieved at 32 dimensions, suggesting that higher-dimensional data is well-suited for parallel execution in R.

- Some irregularities, such as dips at 3D and 16D, may be attributed to suboptimal workload distribution or cache behavior.

- Despite the improvements, the overall speedup remains lower than the one achieved with the C++ and OpenMP version, which reached up to 3.732×, showing more efficient use of parallel resources.

4.2.3 Impact of Cluster Count

Scaling with respect to the number of clusters (k) in R is relatively poor. Performance is significantly lower than that of the parallel K-means implementation in C++ with OpenMP, which achieved a best speedup of 3.823×. In the R implementation, execution times became prohibitive beyond 32 clusters (over 30 minutes), whereas the C++ version handled up to 128 clusters without issue.

Clusters \ Threads	1	2	4
2	1.068	1.547	1.761
4	0.982	2.099	1.648
8	0.984	1.251	1.743
16	0.986	1.428	**2.319**
32	1.096	1.268	1.600

Table 4. Speedup with varying number of clusters.

- Unlike expectations, speedup does not consistently improve with increasing k. In fact, the speedup at $k = 32$ is lower than that at $k = 16$, indicating poor scaling for higher cluster counts.
- The best result is achieved at $k = 16$ with a speedup of 2.319×, but this still falls short of the C++ and OpenMP implementation.
- For small k values (e.g., 2 and 4), performance is limited due to insufficient parallel workload distribution.
- Overall, the R parallel implementation appears to struggle with scaling both at low and high cluster counts, contrasting with the efficient and stable performance seen in the C++ version.

4.2.4 Thread Scaling Analysis

The scaling behavior of the R parallel implementation was evaluated on the AMD Threadripper 2970WX platform. While some gains are observed with increasing thread counts, the overall speedup remains significantly lower than what was achieved using the C++ implementation with OpenMP.

- Speedup increases with the number of threads, showing near-linear scaling up to 8 threads, but begins to flatten beyond that point.
- The maximum speedup observed in R is 17.342× at 48 threads, which is markedly lower than the C++ SoA implementation that reached 26.630×.
- In general, all speedup values are substantially below those of the C++ versions, highlighting the limitations of R's parallel execution model in exploiting high thread counts efficiently.
- Compared to C++, the reduced performance may be attributed to higher overhead, less efficient memory access patterns, and limitations in R's thread scheduling and parallel backend.
- The C++ SoA implementation consistently outperforms AoS, and both clearly outperform the R implementation across all thread counts.

Threads	**1**	**2**	**4**	**8**	**16**	**24**	**48**
Speedup	0.982	1.532	2.712	5.371	7.302	11.240	17.342

Table 5. Speedup results for R implementation across different thread counts.

Impl \ Threads	1	2	4	8	16	24	48
AoS	0.987	1.943	3.811	6.037	8.830	13.640	20.609
SoA	0.998	2.005	3.903	7.861	12.759	19.453	**26.630**

Table 6. Speedup results for AoS and SoA implementations in C++ across different thread counts.

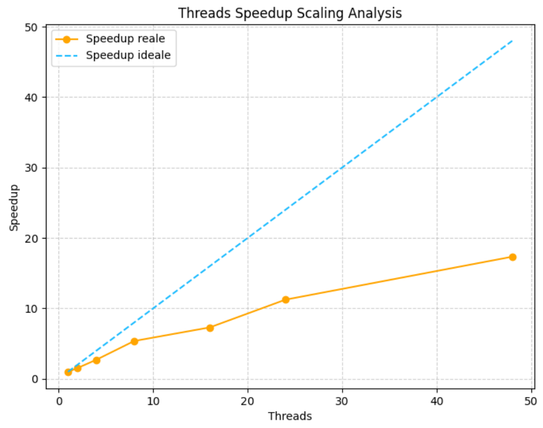

Figure 2. Thread scaling analysis

4.3. Execution Times

The execution time analysis for the R parallel implementation with 100K points, 16 clusters, and 2 dimensions shows notably inferior performance compared to the C++ implementation with OpenMP. Despite processing a significantly smaller dataset, all execution times in R are several orders of magnitude higher.

- For single-threaded execution, the R implementation required 12.64 seconds, resulting in a speedup of 1.088x compared to a baseline sequential run of 13.76 seconds.
- With 2 threads, the execution time dropped to 10.19 seconds, yielding a speedup of 1.489x.
- At 4 threads, the execution time was 6.33 seconds, leading to a speedup of 2.297x — an improvement, but still far from optimal.
- For comparison, the C++ implementation (on a dataset with 1 million points and 32 clusters) achieved execution times as low as 1.2 seconds, demonstrating dramatically better scalability and efficiency even with significantly larger workloads.

These results highlight the substantial performance gap between the R and C++ implementations. While R can benefit from parallelization, its execution model introduces overheads that severely limit its competitiveness in high-performance scenarios. In particular, thread scheduling, memory access inefficiencies, and lack of fine-grained control contribute to much longer runtimes. Even with only 100K points and a relatively small number of clusters, R struggles to deliver execution times within a reasonable range for real-time or large-scale applications.

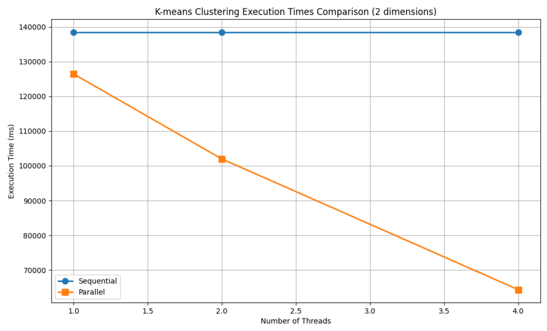

Figure 3. K-means execution times

5. Summary

This analysis highlights the significant limitations of using R with `doParallel` for high-performance parallel programming. While some degree of speedup is observed with increasing threads or dimensionality, R consistently underperforms compared to the C++ implementation with OpenMP across all metrics. Execution times in R are substantially higher, speedups are lower, and scalability breaks down for larger datasets or cluster counts. In contrast, the C++ version handles larger workloads with ease, achieving speedups up to 26x and processing 100 million points without issue. These results suggest that, despite R's convenience for data analysis, it is not well-suited for compute-intensive parallel workloads where performance and scalability are critical.

6. Appendix

6.1. R Source Code Availability

The complete source code for this K-means clustering R implementation is publicly available

and can be accessed through the GitHub repository at:
`github.com/lorenzo-27/kmeans-R`.

6.2. C++ Source Code Availability

The complete source code and the technical report for the K-means clustering C++ with OpenMP implementation is publicly available and can be accessed through the GitHub repository at:
`github.com/lorenzo-27/kmeans-openmp`.